Running Multiple Versions of Services With Continuous Delivery
- Case study at IST

Author: Lucas Wik
Supervisor: Johan Hagelbäck
Semester: VT 2017
Subject: Computer Science
Abstract

Continuous Delivery is a software development discipline where the software is always kept in a release ready state. It has proven to be a challenge for companies to adopt the practices of Continuous Delivery, but the benefits it brings may well be worth overcoming the challenges the adoption process brings. But the problem with the challenges is that they appear to be unknown, different adoption cases report different problems, some even consider something a solution whilst another case considered the same subject to be a problem. Thus adopting Continuous Delivery is a tricky process. But the tech company IST is interested in adopting Continuous Delivery and are looking to take a soft start by adding new functionality to their service-oriented system, this functionality is to be able to run multiple versions of their services at the same time. This study has implemented this functionality in their system and then researched possible issues or benefits the functionality would have towards Continuous Delivery. Finally, a discussion was made on how the author thinks approaching Continuous Delivery should be done by any company or developer interested.

Keywords: Continuous Delivery, running multiple versions, Service-oriented architecture, Consul, Consul Template, Nginx, Docker, Container
Preface

I want to give thanks to the supervisor of this study, Johan Hagelbäck, as well as the other students who peer reviewed this work. I also want to give big thanks to IST as a company for the cooperation for this study and Andreas Törnström at IST for the assistance I received throughout the study.
Contents

1 Introduction 4
   1.1 Background 4
      1.1.1 Continuous Delivery 4
      1.1.2 IST 5
   1.2 Previous research 5
   1.3 Problem formulation 7
   1.4 Motivation 7
   1.5 Research Questions 8
   1.6 Scope/Limitation 8
   1.7 Target group 8
   1.8 Outline 8

2 Method 9
   2.1 Scientific Approach 9
   2.2 Method Description 9
   2.3 Reliability and Validity 10
   2.4 Ethical Considerations 10

3 Implementation 11
   3.1 Registering Versioned Services In Consul 11
   3.2 The Dot Problem 14
   3.3 Implementation Rules 16

4 Results 18
   4.1 Implementation Result 18
   4.2 Implementation Tests 19
   4.3 Database Updates 22
   4.4 Pros And Cons Of The Implementation 24
   4.5 Comparison Towards CD Research 24
   4.6 Analysis 27

5 Discussion 29

6 Conclusion 33
   6.1 Future work 34

References 35
1 Introduction

Continuous Delivery (CD), not to be confused with Continuous Integration or Continuous Deployment, is the practice where the code in a software product is always kept production ready [1]. It allows for more frequent deployments which reduces the risk when deploying new code compared to doing big deployments. And in turn, doing more deployments with CD yields faster user feedback [1]. For these benefits, companies such as IST is interested in adopting CD. IST is looking into adopting CD with some new implementation that is to run multiple versions of their services. This study will be implementing and testing the functionality IST is after, then comparing it against existing research about adopting CD to identify how well it would work with CD.

1.1 Background

This study will cover two main subjects, the implementation at IST and CD. Therefore these two subjects be discussed individually regarding their background.

1.1.1 Continuous Delivery

Continuous Deployment and Continuous Delivery are known to be used as synonyms, but Continuous Deployment is an extension of Continuous Delivery [2]. As stated, Continuous Delivery follows the principle of keeping the code production ready [1]. Continuous Deployment, however, follows the principle of constantly pushing the code to the production environment [3]. The difference is that Continuous Delivery is ready at a moment’s notice to make a new release while Continuous Deployment constantly commits to the production environment.

According to M. Fowler, there is a set of principles that needs to be followed for using CD [1]. These principles are:

- Your software is deployable throughout its lifecycle
- Your team prioritizes keeping the software deployable over working on new features
- Anybody can get fast, automated feedback on the production readiness of their systems anytime somebody makes a change to them
- You can perform push-button deployments of any version of the software to any environment on demand
By following CD and its principals M. Fowler also suggests its benefits. First, it reduces the deployment risk because the software can be released more often which results in smaller changes for each release [1]. This also creates the next benefit, faster user feedback [1]. Since you get your working software to the customers faster, you will also receive feedback faster. Lastly, M. Fowler also suggests that it provides better visibility of progress since you do a lot of releases [1].

1.1.2 IST
IST offers administrative systems, applications for learning and much more to schools in Scandinavia. In their new platform, they are not happy with how they are handling updates to their software. At the moment they are doing big updates every 3rd week or so. Therefore they want to implement support to run multiple versions of their services in their system. With this, they would be able to do updates more often and with less risk. If a new version would contain critical bugs or does not perform as expected it should be easy to rollback and use a previous version of the service instead.

IST has provided some examples for desired functionality, these are:
“/person/3.2”
This request would use version 3.2 of the person service.
“/person”
This request would always use the latest available version of the person service.

With this functionality, IST plans to take another step towards adopting Continuous Delivery.

IST has a service based system using Docker containers for holding each service and Consul as a service discovery tool. Nginx with Consul Template are used as a load balancer for their services. IST suggested that these technologies and components will most likely be used or require modification to achieve the desired functionality.

1.2 Previous research
There are a lot of studies about CD, many of which are case studies for adopting CD because the adoption process has proven to be a big issue.

E. Laukkanen, J. Itkonen, and C. Lassenius wrote a study about CD where they summarized a lot of adoption cases, the issues those cases faced and the solutions they used [2]. Besides providing a lot of summarized
information, their study also proves that adopting CD is something that should be researched first before executed (which is also recommended in their conclusion) [2]. This is because the cases they reviewed did not all report the same issues, thus each case can face different issues. Some cases might also share similar problem areas but they might have different issues in those areas. An example of this is when they discussed the problem of having an unsuitable architecture in their study [2]. One case reported this to cause time-consuming testing while another case reported it to cause problematic deployment [2]. These problem areas are something that they identified in their study, and each problem discussed was categorized in a problem area. The identified problem areas are Build design, System design, Integration, Testing, Release, Human and organizational, and Resource [2]. These areas provide good starting points for identifying benefits and hindrances of the implementation.

Since the implementation will be under so many specific requirements (language, frameworks, tools and so on) there is no previous research relevant to the implementation itself. The implementation, however, is somewhat similar to blue-green deployment [7]. Blue-green deployment is to run two versions of your environment, one you call green and the other one blue, then you pass all of your traffic the stable environment while preparing the next version on the other environment [7]. An example is if the current stable version is the green environment, the next version goes up on the blue environment. Whenever the blue environment is considered ready to be released all the traffic now moves to that environment instead, but the green stays the same even though inactive [7]. This way, if the blue environment introduces bugs or issues, all the traffic can go back to the green environment while fixing the blue version for a re-release. When the blue environment is stable, the next version is built in the green environment and the process restarts [7]. This can be related to the implementation that will be made in this study, but the difference is that multiple versions of a service will be active at the same time in the same environment, not have multiple environments active (or ready to be activated).

Klokan technologies, a Swiss software company, wrote a short blog article about their use of blue-green deployment while also using similar technologies that IST uses (Docker and Nginx) [8]. This article exemplifies a simple program that uses blue-green deployment with these technologies, this does not provide good guidelines for the implementation of this study but it
builds confidence that the implementation is possible, given the similarities between blue-green deployment and the implementation for this study.

1.3 Problem formulation
The first step for this thesis is to create a copy of IST’s architecture where the functionality to run multiple versions will be implemented. The copy will contain as minimal functionality as possible to still be regarded as a copy, this meaning that the same technologies and principles will be used. With the new functionality in the copy, pros and cons of the implementation can be identified and exemplified. What is regarded as cons in the implementation is things or effects that would lead to possible problems. An example would be something like writing to a database. If two different versions would write to the same database and same field at the exact same time, how would or could this be handled? These cons, however, can have solutions that would reduce or remove the problem. Pros are the pure opposite, things or effects of the implementation that would have a positive effect on workflow, system functionality, and more.

Using the pros, cons, and functionality of the implementation and then comparing them to identified problems and solutions from existing research for adopting CD, the implementations benefits or downfalls for adopting to CD can be identified.

1.4 Motivation
The goal of the implementation of running multiple versions is to be implemented into an existing project. This is the main reason for identifying pros and cons regarding the implementation in the copy of the architecture. The more pros and cons identified, the easier the implementation would become in the existing project.

As described in the background section, CD provides a lot of benefits, but that does not mean it is for everyone. Adopting CD has proven to be a challenge for companies, and since each adoption case differs from the other, no step-by-step guide can be set in stone for a successful adoption process [2]. Therefore studies like this one need to be made to build confidence in the adoption process as well as providing new possible problems or risks for future cases. The paper written by E. Laukkanen, J. Itkonen, and C. Lassenius summarizes a lot of general problems and solutions other adoption cases have faced, but different cases can consider different things to be problems [2].
1.5 Research Questions

<table>
<thead>
<tr>
<th>RQ1</th>
<th>What pros and cons can be identified when running multiple versions of services in IST’s system?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ2</td>
<td>How well would the implementation work with Continuous Delivery?</td>
</tr>
</tbody>
</table>

The result for RQ1 is expected to provide the cons and pros in the implementation that has the most effect. Solutions or suggestions to handle the identified cons is also expected to be included.

The result for RQ2 is expected to compare the implementation towards the problem areas described by E. Laukkanen, J. Itkonen, and C. Lassenius in their research [2].

1.6 Scope/Limitation

An important note is that it is only the implementation that is going to be tested towards CD, not the company as a whole. What this means is that something like version control might come up as a con or pro for the implementation, but it will not be compared or discussed towards how IST is currently working with version control. This would simply make the project too large for its allowed time frame.

1.7 Target group

The main target group for this thesis is IST. But it is also for anyone interested in similar implementation or adopting to CD.

1.8 Outline

The following contents of this study will be the Method chapter describing the methods and plans to generate a result, the Result chapter describing the results from the executed method, the Discussion chapter with the author's discussions regarding the study its results and finally the Conclusion chapter summarizing the results and discussions.
2 Method

In this chapter, the approach for answering the questions, the validity and reliability of the approach, and the ethical considerations with IST will be discussed.

2.1 Scientific Approach

This is a qualitative case study at IST to further build upon the current research for adopting CD.

IST has provided guidelines and suggestions for what technologies and programming language to be used for creating the implementation. These suggestions and guidelines will be followed provide a more satisfactory result for IST regarding the first research question.

The second research question will answered with a short literature study where the implementation pros and cons will be compared towards existing research about adopting CD to identify how well the implementation would work with the principles of CD.

2.2 Method Description

This paper is going to analyze the implementation of running and supporting multiple versions of services in IST’s system. IST store their services as Docker containers on Rancher, they use Consul as a service discovery and Nginx as a load balancer with Consul Template to dynamically modify the Nginx configuration based on the services registered in Consul. The architecture copy will be a given Rancher environment IST has provided for creating and testing the implementation. As suggested by IST, Java applications using Maven and Spring with SpringBoot will be created to be used as “dummy” services in place of actual services. One of these Java applications will have a database relation to a locally hosted Oracle database 11g Express edition, this database was requested by IST to use for testing. Other “dummy” applications in other programming languages might be created if required further on.

The “dummy” components will call each other in the same manner IST is currently doing with their services, but with version modification. With this, the implementation could be considered complete.

When the implementation is working as expected, it can be analyzed for pros and cons. The process for identifying pros and cons will mostly
consist of purely testing the implementation to prove the status of an identified pro or con. The identified cons might lead to further implementation modification. When the implementation has been thoroughly analyzed, the identified cons that do not have a complete solution can be compared towards existing research for adopting CD. Doing this might provide possible suggestions or solutions to handle the problem. Comparing the implementation towards existing research can also provide insight on cons not yet identified. The comparison process will mostly be looking for keywords or key-features a con is related to and then comparing it towards an existing identified problem that shares similar keywords or key-features. To identify new possible problems regarding the implementation, keywords or key-features related to the implementation itself will be looked into. When that process is finished, a conclusion can be made about how suitable the implementation is for IST and new possible adoption problems for CD might have been identified.

2.3 Reliability and Validity
The reliability of the research is quite high. If using the same technologies and doing the implementation in a similar structure, one should reach similar conclusions regarding the implementation. Same goes for when comparing the implementation towards CD if the same sources are used similar conclusions would be reached.

The validity of the research is not too great based on that the writer is a student and an experienced developer would have greater chances of identifying possible cons or pros of the implementation. Though the most critical pros or cons are expected to be covered.

2.4 Ethical Considerations
This study will not be giving out any information that IST does not want to be shared. But since it is mostly just the same technologies in a structure with the same principles as their system, they are not all too restrictive of what can be discussed.
3 Implementation
This chapter will describe the implementation that was executed at IST. Since the author of this study did not have a lot of prior experience with the technologies used nor about how they were used at IST, the main goal became to first handle the registration in Consul and then move on to any problems that would occur since there was not enough knowledge to predict possible problems. A lot of different approaches were tested to achieve the desired functionality but only the final result will be discussed in this chapter.

3.1 Registering Versioned Services In Consul
As previously mentioned, IST uses Consul for service discovery and Docker to containerize their services. To register a service, a container needs an environment variable named SERVICE_HOSTPORT_NAME set within its scope, the value for this variable should represent the name of the service. For example, the environment variable ‘SERVICE_8080_NAME=person’ would register the service ‘person’ and tell the system the service is hosted on port 8080. This environment variable can be set in the dockerfile or during startup for a container, as long as it exists within the scope of the container it will be registered in Consul as a service. The registration process itself is made through a registration application they have written in Node.js. Since all of the services that will be registered in Consul goes through this registrator application, the logical option became to implement the version handling for each service in this application to create an easy implementation process.

![Diagram](image-url)

Fig. 3.1 Exemplifying the flow from starting a container in the Rancher environment to the registration of the service in Consul.

To be able to handle the version modification in the registrator, some form of version information needed to be passed with each container. This
was achieved by creating an environment variable named VERSION for each container. Where or when this environment variable is set does not really matter as long as it exists in the scope of the container and holds the version value for that service. At IST they use a version structure of major > minor > revision, e.g. 1.5.4 (1 is major, 5 is minor and 4 is revision). Important to note is that an increase to the major version value is made when breaking changes are made, thus very important for API consumers. An increase to the minor or revision version values, however, should not cause breaking changes for API consumers. Since the system is service based, a service that requires the use of another service handles that communication through RESTful API’s built into each service.

For this implementation, the VERSION environment variable was set in the Dockerfile and given the version value from the pom.xml file or manually set, but it can also, for example, be set in a docker-compose.yml file or during the docker run command. Since each service is containerized by docker, each container's environment variables will only be available from the scope of that container.

When starting to modify the registrator application, to make sure that the services that will not use the version modification will still be treated the same way as they currently are, values (like ‘version’) were added to the service objects in the registrator for services that have the VERSION environment variable defined. This way, if these extra values for version modification are not defined, the registration process will execute as previously. Below are pseudo code examples, 3.1 and 3.2, where services get filtered out if already registered, 3.1 shows how it is currently done with their live version, 3.2 is with the version modified filter.

Previous

```plaintext
If Service is not currently registered
    Register the service
endif
```

Code 3.1: IST current filter for filtering out already registered services

New

```plaintext
If (Service does not have version value defined AND it is not currently registered)
    OR (Service does have version value defined AND <requirements for passing a version modified service>) then
    Register the service
endif
```

Code 3.2: Implementations filtering of already registered services
The next step became adding the version value to the service name and then registering it in consul. This was made after passing the filter for already registered services. If the “version” value for a service object is defined, the service is assigned another value called “group”. The “group” value is a shared value that all versions within a certain service has, for example, if a service called “person” is getting registered as version 2.2, then this service will be registered under the name “person/2-2”, but the group value will be set to “person”. This is to allow easy detection of what services are the within same core group of a service to handle the next task, creating a service that goes to the latest version of a service in a group.

In the registrators starting process, an event is created to run every 30 seconds to synchronize the containers with the Rancher API. The decision was made to also synchronize what version of a service would be used when requesting its service group value (e.g. “/person”) during this process. As mentioned previously, this is requested by IST to always default using the latest active version of that service, so if the latest active version of the service “person” is 2.2, the requests “/person” and “/person/2.2” would yield the same result. Since each service with a version value added also have its group value defined at this point, these group values can be used to distribute each service into a list with all of the other services within the same group. Then, any active services registered under a group name can be compared towards the existing group values, if there are no active versions within that group, the service registered under that group name gets deregistered.

With all of the services split up into lists of their service groups, the next step is to compare all of the versions within each service group to identify the latest version. A function was made to compare the versions between two service objects, the code example 3.3 shows the core concept of this function. Note that the entire method is not shown, if any of the given service objects are null, the other object is returned (if they both are null an exception is thrown), and if the values would be equal, then null is returned.
Code 3.3: Illustrates how the latest version is identified with a possible infinite amount of version values.

After iterating all of the service objects through this method to find the latest version, that service object is compared again against the current latest version, and as long as null is not returned the returned object is set as the latest version. There is no overwrite process when null is returned because, as stated earlier, then the versions are equal and when changing what version of a service that is used by a service group, a small interference will occur since the nginx.conf file will be updated and reloaded. A final note regarding the implementation in the registrator is that the service under a group name leads to the same instance of the latest version, it does not have its own instance.

3.2 The Dot Problem

As can be seen in the 3.1 segment, the versioning is written as X-Y and not
X.Y at certain places. This is because services get registered with their names this way, so a service called ‘person’ getting registered under version 2.2 ends up being registered with the name “person/v/2-2” in Consul. This was due to a problem identified regarding dots in service names after registering a service like “person/2.2”. Before describing the solution to the problem, it is important to know that Consul does not support the use of dots in service names, dots are allowed but warnings are displayed if used [9]. This is because dots in service names interferes with their DNS interface for querying services [9]. The only reason Consul allows dots in service names is because they know that some people rely on it so they can not disallow the use of dots in service names as of this writing [11]. But the main problem was not with this interference in Consul, IST does not use their DNS interface, the main problem was with Consul Template that does not allow service names to contain dots [10]. To give a short overview, the code example 3.4 is a typical way to write the upstream block in the nginx.conf file for each service registered in Consul using Consul Template.

```
{{ range services }}
  upstream {{.Name}} {
    {{ range service .Name }}
      server {{.Address}}:{{.Port}};
    {{else}}
      server 127.0.0.1:65535; #force a 502
    {{end}}
  }
{{end}}
```

Code 3.4: A general execution with Consul Template to create the upstream block in the nginx.conf file.

The row that caused the issue is the third one, `{{ range service .Name }}`. This queries Consul for a service registered with the value of “.Name” and loops all of the containers in that service and creating the server block for each container with its IP-address and port number. But if the “.Name” value contains a dot (or any other illegal characters) when doing executing this, an error is thrown and the Nginx load balancer crashes. No identified way to bypass this was concluded in this study, however, the server block does not define the endpoint for the API, this is defined by the location block in the nginx.conf file. Thus the functionality could still be achieved by just modifying the endpoint instead. How this was made is shown in the example below.
Changes the API endpoint for a service so the ‘-’ character is replaced with a ‘.’ character in the version string.

To describe the above example in more detail, it first checks if the service name contains “/v/” since this value is added to all services with version modification. Then, if it does contain “/v/”, the “X-Y” version value is extracted into the $ext variable, the dash is replaced with a dot and then a slash is added so the value of $ext becomes “/X.Y” where X and Y are the version values. The name of the service up to the ‘/v/’ string is then extracted and merged with the value of the $ext variable which creates the endpoint “service/X.Y”. With this in the nginx load balancer, a service registered as “person/v/2-2” will have the endpoint “/person/2.2/”. It is important that the version value is extracted and then the “-” characters are replaced with “.” characters, otherwise this would cause issues for any service names with a “-” character in it.

3.3 Implementation Rules

The new implementation has a set of rules to be followed for proper use. If these rules are applied into their existing system with the updated load balancer and registrator, it will work as presented in this study. The following rules need to be applied to each service:

- Define environment variable VERSION for any service that should get versioned, the value should represent the version of the service. The version can contain any amount of version values separated by dots or dashes (dots are replaced with a dashes in the registrator).
- The value for VERSION can not contain any strings like “beta”.
- The string “/v/” may NOT be used in service names, this would cause unwanted executions in Consul Template.
- The endpoint for a version modified service is not the same as the
service name, the “/v/X-Y” part should be translated to “/X.Y”.
- If a service, such as “person”, has a version modified service registered, a service of “person” should not be started without version modification.
4 Results

This chapter will cover the result of the implementation and the results from the tests executed to the implementation. Identified pros and cons regarding the implementation will be presented as well as results regarding how well the implementation would work with CD based on existing research.

4.1 Implementation Result

The requested functionality by IST was successfully implemented into a given Rancher environment. The implementation was executed by modifying the Consul registrator and load balancer applications running in the Rancher environment. Fig. 4.1 shows how the Consul UI looks with version modified services.

![Fig 4.1 Consul UI with the modified registrator application.](image)

In fig. 4.1 the viewable registered services are “dummy” components created by the author. For testing the implementation three of these “dummy” components were created for different forms of testing the implementation. ComponentA (not displayed in fig. 4.1) was used for simple testing. It is a simple SprintBootApplication for executing small tests like checking if the service was reachable, the API endpoint worked as expected and when
testing the registration in Consul, this component was not used for the later stages of testing where more specified components were created. ComponentB, also built as a SprintBootApplication, was created for testing the Oracle Database Express Edition v.11g. A simple database table was created in the Oracle database and ComponentB contains simple CRUD functionality towards this database table. The last component, ComponentC, is a Node.js application used for testing database requests in ComponentB. ComponentC was written as a Node.js application to be able to make asynchronous calls to ComponentB, this is something that is easier done with Node.js compared to Java. The services displayed in fig. 4.1 can be reached as following:

- ComponentB version 1.5 can be reached via the request “/componentb/” or “/componentb/1.5/”.
- ComponentB version 1.0 can be reached via the request “/componentb/1.0/”
- ComponentC version 1.0 can be reached via the request “/componentc/” or “/componentc/1.0/”.

To use the implementation, a set of rules needs to be followed as described in subsection 3.3 in the previous chapter.

4.2 Implementation Tests

The main goal to achieve a successful result of the implementation is to pass the following requirements:

1. The request URL provides the response from the expected requested version. Example is that when requesting version 1.0 you get the response that is expected from version 1.0, not another version.
2. Requesting a service group name results in requesting the latest version of that service.
3. The service under a service group name should always lead to the currently latest version within that group. It should be re-set whenever the current latest version is taken down or becomes inactive or if a newer version is booted.
4. When there are no active versions of a service, there should be no service active under that service group name.
5. Simultaneous requests executed to two different versions for database update requests should not cause critical errors. Critical errors would be anything that would cause a crash or leave requested database
queries unexecuted.

Requirement 1 and 2 were exemplified and proved in the previous subsection. Requirement 3 and 4 were tested by running two versions of ComponentA and when requested it returned the version of that instance. The table below, fig. 4.2, displays the result of this test, each row represents a new test but during all of these tests no new updates or uploads were made to the Rancher environment. Any difference in active versions between each test just means that a version was stopped or started in the Rancher environment. These results prove the demands of requirement 3 and 4.

<table>
<thead>
<tr>
<th>Test Description</th>
<th>Request</th>
<th>Active Versions</th>
<th>Expected</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiate</td>
<td>/componenta/</td>
<td>1.0 and 1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Remove lower version</td>
<td>/componenta/</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Add lower version</td>
<td>/componenta/</td>
<td>1.0 and 1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Remove current latest version</td>
<td>/componenta/</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Add higher version</td>
<td>/componenta/</td>
<td>1.0 and 1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Remove all active versions</td>
<td>-</td>
<td>-</td>
<td>No service under the name “componenta” active.</td>
<td>The service “componenta” was deregistered after the registrator sync.</td>
</tr>
</tbody>
</table>

Fig. 4.2 Showing the result from different requests.

The last requirement 5, collisions needed to occur when requesting updates to ComponentB to see what would happen. As described in subsection 4.1, ComponentC is used to spam requests to ComponentB to
cause several collisions. Two versions of ComponentC would send 80 requests each to their own versions of ComponentB. 80 requests was the amount decided on the basis that when the requests would exceed over 200, there was a risk that it would overload the hardware. Executing 80 update requests took on average 10 seconds to process, more than enough time to execute the command on two different versions to cause multiple request to collide.

The first test was too see how the database would act when only one version of ComponentC spammed one version of ComponentB. This was tested first to be able to see how the database reacted when getting spammed by 80 requests so that any differences could be spotted when two versions of ComponentC would be spamming two versions of ComponentB. The result ended up being that it could handle all of the requests, but they were not executed in the same order as they were requested. Each request had a number assigned to it, 0 to 79, which was used to update the column value. The first request would be to update the column value to 0, the second one to value 1 and so on until 79. Since these requests were initiated asynchronously they were not expected to be received by ComponentB in the same order they were initiated, 0 to 79. Fig. 4.3 exemplifies this (120 requests were used when only spamming one version).

![Fig. 4.3 Showing that requests were not received the same order initiated.](image)

What was not expected though was that the requests were not finished in the same order they were received which is illustrated in Fig 4.4. The left part shows the first received requests, the right part are the first finished updates. As shown in the image, few numbers that were received first got processed first.
Fig. 4.4 Showing how the requests were not executed in the same order they were received.

Into the reason for this, it is most likely due to how Java handles multiple threads. Each request initiates its own thread, and the threads in Java execute their task whenever they can acquire process power to do so, thus the order of thread initiation does not guarantee the order of execution. But there were still no problems to initiate multiple connections to the database.

After these first tests, two versions of ComponentB and ComponentC were started and the same functions were executed. The only notable difference was that the execution time for the requests to be processed was almost doubled. No critical errors occurred for updates, thus requirement 5 is passed.

4.3 Database Updates

A possible problem that comes to mind when running multiple versions is database upgrades. Graham Brooks, the technical principal at ThoughtWorks, wrote a blog post about how database upgrades can be done with BlueGreen deployment, which is relatable to this implementation [13]. The idea is to have support for two versions, both old and new, at the same time. He provides an example of how this could work when adding a new table to the database, fig. 4.5 to 4.7 illustrates this process.
Figures 4.5 to 4.7 shows how an update to the database could be processed. To summarize, fig. 4.5 would be the current version, then the decision to add the “phone” table gets decided. A new version of the software is built towards working with the new table and not the old values, but the old version can still be used since the values it uses are not removed (shown in fig. 4.6). After the implementation has been tested thoroughly, the data used for the old versions can be removed as shown in fig. 4.7, but in turn the old version that are not adapted to the new database will have to be updated or also removed.

Figure 4.6 and 4.7 illustrates how an update that includes removal of data can be made. A new version is made that only uses the data that is not to be removed, then after testing the new version the unused data and the old versions can be removed.
4.4 Pros And Cons Of The Implementation

The main problem with the implementation is the possible miscommunication regarding the service name for a version modified service and its API endpoint. Since all other services registered have their endpoint set to that of its service name in Consul, it is very possible that someone at some point will request the wrong endpoint and get an error without realising why. A solution would be to have the same endpoint as in the service name (e.g. “/person/2-2/” as the API endpoint) but the current versioning standard at IST is by using dots separating major, minor and revision version values.

Another problem with the implementation is that it does not support itself. Only one version of the registrator and load balancer can be active at a time, thus if any updates to these components is needed a more complicated process would also be required to execute a rollback. The same is relevant for any services not using the version modification. The core of this issue is that doing a rollback for certain components require different processes, thus it is not a complete solution to possible issues after a new deployment compared to a BlueGreen deployment. This is, however, also the main pro of the implementation, easy rollback functionality for the services using the implementation. An example would be if a new version of a service is published and the previous version is still active, if the new version would cause issues it can just be taken down and after the synchronization process the service group will be modified to lead to the previous version that is still active in the environment. Besides an easy rollback functionality for a broken version, it can also be used in the same way if a new feature gets bad feedback from the users and a rollback is made to the previous version to satisfy the customers.

The last notable benefit of the implementation is version specific requests, meaning that if a service relies on an older version of another service it can just make a modification to the request URL to use that older version. This assumes that the older version is running on the environment.

4.5 Comparison Towards CD Research

This comparison process towards CD will use the study written by E. Laukkana, J. Itkonen and C. Lassenius, any of their identified problems that are relatable or countered by the implementation will be described and
discussed. A problem can also be placed as ‘mixed’ meaning that it is a problem that can be both countered and relatable.

**Countered problems**

**Broken build**
This problem is described to occur whenever a new build of the software is ends up broken and requires significant effort to be fixed. This problem is countered since the implementation can simply do a rollback if a new version of a service ends up broken.

**Database schema changes**
As described in subsection 4.3, database schema changes can become easier if there is support for multiple versions of the software since you can support a known functioning version while testing a new version.

**Deployment downtime**
When working with CD, more deployments are made since the code is constantly in a production ready state. The problem here is that more deploys also results in more deployment downtime. This is, however, reduced with this implementation since when updating or uploading a new version, an old one can run in its place until the new update is finished.

**More deployed bugs**
This problem relates to that when using CD more deployments results in more deployed bugs. This is not an issue with the implementation since an easy rollback to an older version can be made, thus in turn handling any newly released bugs.

**Third party integration**
Third party integration is reported as a problem due to that frequent releases complicate the use of the software for a third party. But with the versioning, any API for third party users could be versioned and allow easier and more reliable API consumption for third party users. This problem, however, is not related to IST at the current moment.
Relatable problems

Time consuming testing
Since more versions are kept production ready, more tests must be executed since the versions differ from one another.

Team coordination
Keeping many versions of a software can make team coordination more difficult. Knowing which is the latest version, what versions that have a certain feature, and so on, makes more effort required for keeping a good team coordination.

Documentation
Documentation is a problem when the documentation does not represent the correct version. This was reported to be caused by people with low experience and do not follow industry standards. This becomes even harder when supporting multiple versions since non-updated documentation could cause severe problems when using multiple versions at once for a service.

Mixed problems

Work blockage
This problems relates to whenever a development task is delayed or blocked due to broken build or other integrations in a queue. To exemplify how the implementation can be a problem and a solution in this regard, imagine these three different services: Service_a, Service_b and Service_c. Both Service_a and Service_b depend on Service_c, the team for Service_a is ready to use an upcoming change in Service_c but the team for Service_b is on vacation and cannot adapt to the upcoming change. Service_a on the other hand needs the change in Service_c to proceed with their tasks. In a case like this, Service_c can have two versions active, one that Service_a needs and another one that Service_b needs. This example would solve the problem. Service_a can however still be blocked by Service_c if it does not yet have the feature that Service_a needs, thus it can still be a problem. The problem, however, is mitigated by the implementation.
Long-running branches
This problem is described to be due to merge conflicts occurring when developing on a branch for a long time. Though it is not necessarily the case for this implementation since keeping a branch is mostly to be able to publish its version and not to develop on it. What is more relevant about this topic is that branches would still persist and be relevant for a longer time, or as long as support for that version is wanted. The problem in itself would be if any new problem would arise in an older version and having to go back to old code. This is however not necessarily a problem since this in turns allows for the implementation to be easily handled and if a problem would arise in an old version it can just be removed instead of updated.

Broken Development flow
This problem relates to when failures occur which distracts the developers and slows down the productivity. It is described to be caused by the problems ‘Broken Build’ and ‘Time-consuming Testing’. ‘Broken Build’ has been described to be a countered problem while ‘Time-consuming Testing’ has been described to be a relatable problem, thus leaving this problem as mixed.

4.6 Analysis
During this study, it has become clear that trying to identify any new possible problems the implementation can have towards CD that are based on facts is impossible. This is most likely due to a lot of the variables that adds up for a successful CD process is within an organization itself. BlueGreen deployment, something that is regarded to work well with CD, can still fail if, for example, the developers do not follow standards and the principles of CD [7]. This means that if even something designed to work well with CD can fail to create a successful CD process, it becomes hard to state what will go well and what will not for a CD process based on assumptions or ideas. There is no denying that a certain implementation or functionality, like BlueGreen deployment, can work better with CD than something else, but at the same time there is no guarantee to say that it will work well. It merely becomes a way to give more control and idea for how it should or could work. The same goes for this implementation, since no actual adoption process has taken place, no guaranteed conclusions can be made for an adoption process using this implementation. The only way a comparison can be made towards CD is with existing research from previous adoption cases.
Regarding the implementation itself, the results are satisfactory. This is purely due to the goals for the implementation being reached, but it is still an annoyance that the services in Consul could not have a dot in their names. The implementation could also have been tested more thoroughly since only “update” and “select” queries were tested. It could also have been a lot easier to analyze the query results if, for example, one version only received even numbers and the other version only received uneven numbers as update values. This would make it easier to identify whenever a collision between the two versions occurred for the “select” queries (if the version only updating even values received an uneven value for the select query after the update query there would be an obvious collision). This was noticed in the current tests but it was an annoyance to discover.
5 Discussion

This chapter will be discussing the result from the research with the author's opinions and conclusions.

RQ1: What pros and cons can be identified when running multiple versions of services in IST’s system?

Several pros and cons were identified regarding the implementation. The main issue being that services do not have the same API endpoint as their service names in Consul. The possible solution to this would be to split version values with some other character than a dot, but this would go against the versioning standards at IST. My personal opinion is that changing their versioning standard is not worth the effort considering the size of the company (hundreds of developers as of current).

The second problem is that the implementation does not support itself, which is quite critical. With this implementation the registrator and load balancer applications will become more critical parts of the system than they already are since they get assigned more responsibility. Thus if this implementation would be used, it is highly recommended that the load balancer and registrator applications are tested and confirmed stable before using the versioning functionality. When starting to test it, only a few services should use it. This is because if any problems would occur when services are adapted to using version specific requests, doing a rollback of the load balancer and registrator will require more effort since the services that adapted to using the implementation would also require a rollback (e.g. a service modified a URL to request a specific version).

The main pro of the implementation, in my opinion, is the rollback functionality. It can result in so many benefits such as more reliable releases, easier update process, easier to downgrade a service, a better development flow, and more available functionality. These are the benefits I could list, an inexperienced developer, so it is safe to say that the functionality to easily upgrade or downgrade what version of a software program is used can result in many more benefits in all kinds of projects.

There were also some rules established to use the implementation. Some of these rules could be considered as cons, for example the version environment variable cannot contain any strings such as “beta”. This issue, however, can be fixed when comparing two versions but to fix this was not
requested by IST when the issue was presented. It might be something that will require more discussion before a decision can be made by the company. The registrar will not crash if a version value contains a string, but the comparison between the version values will become invalid and unreliable.

The rule that the service name may not contain “/v/” is in my opinion not a con since it is unlikely that it will restrain any service names, but it is an important rule nonetheless.

Last is the rule that a service that has a version-modified service registered should not have any service registered with no version-modification, e.g. if “person/v/2-2” is registered and a service named “person” is started without the VERSION environment variable. This will cause severe issues with unexpected behaviour since the service that will be registered under the group name of “person” and the non-version-modified service “person” will collide in the registrar. This was never tested due to being discovered just recently, which is troublesome, but the assumed outcome is that if a service under the group name of “person” is registered and then if a non-version-modified service of “person” tries to get registered, the non-version-modified service of “person” will get denied due to the service ID already existing, but there is no proper error message for this conflict. If a non-version-modified service of “person” is registered and a version modified service of “person” is then started afterwards, then the group service should overwrite the non-version-modified service “person” at the next sync.

**RQ2: How well would the implementation work with Continuous Delivery?**

There is not much to discuss regarding the CD problems that were compared towards in the Result chapter. The presented result was based on logical connections between the descriptions of the problems and possible relations with the implementation. There are, however, problems worth discussing why they were not regarded as relatable issues. These are presented below.

*System modularization*

One can consider modules and services to be relatable enough to have this issues relatable as well. But the description of this issue is that system modularization causes effects that in turn causes problems or solutions. Thus since it is specifically towards system modularization and not a service based
system, this problem was decided to not be relatable enough.

**Internal dependencies**

This issue is more relatable towards system architecture and not the implementation itself, and if anything, this problem is countered since internal dependencies can be more easily managed by supporting more versions of a service. Or it can be relatable due to the heavy dependency on the registrator and load balancer applications.

**More pressure**

When you have a reliable rollback functionality it is a possible argument that some pressure would be relieved on the developers. But the relatable problems presented in chapter 4 needs to be backed by logic and reason for its status, not arguments. The problem is also described that there is more pressure on the developers when the code is required to always be in a releasable state, thus also not very relatable to the implementation itself and more related to the developers.

The comparison process provided the result that there were more identified countered problems than relatable ones. Thus overall, the implementation could be considered to be viable for CD based on this comparison towards the study written by E. Laukkanen, J. Itkonen and C. Lassenius [2]. As described in the analysis of the result, identifying new possible problems the implementation could have with CD is impossible from my point of view. There is not enough knowledge to be able to say or predict possible problems with reason. This is mostly because I have not worked with CD during my studies, only read about it. A person with experience for adopting CD would be a much more reliable source for any possible unidentified problems. Simply reading about CD did not give enough insight to give confidence in predicting unidentified issues.

To further discuss that simply reading about CD is not enough to predict possible issues with CD, one can look at the points Martin Fowler states that describes the requirements of CD [1].

“Your software is deployable throughout its lifecycle” and “Your team prioritizes keeping the software deployable over working on new features” Very developer dependent, is not very affected by a form of functionality or
implementation.

“Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them”
Very dependent on architecture and how the organization works.

“You can perform push-button deployments of any version of the software to any environment on demand”
This one is relatable to the implementation, but only half of it regarding the versioning. How the deployment is processed and deploying with the push of a button is not relatable to the implementation at all.

All of these topics relate back to one thing, experience. The experience of what is regarded as deployable code and what it means to constantly keep it deployable, the experience of how feedback can be communicated between teams in a clear and effective way, and experience of creating a deployment pipeline with reliable versioning. So to predict any new possible problems in any of these topics without any experience is not reasonable in my opinion. E. Laukkanen, J. Itkonen and C. Lassenius also states in their study that system design problems are the most common ones and are causes for multiple other problems with few possible solutions, thus it is very important to have a developer experienced with CD looking at any form of system design to have possible predictions of problems so that they can be handled or be prepared for before a CD adoption process [2].
6 Conclusion

CD is an ever growing interest for companies to adopt. The process to adopt it has proven to be a challenge and thus larger companies with multiple teams want to prepare for the adoption process in any way possible. IST is one of the companies that are looking into adopting CD and to start out they want to be able to run multiple versions of their services that they register in Consul. In this study, a solution for this functionality was implemented into existing components in IST’s system and the following questions were answered.

RQ1: What pros and cons can be identified when running multiple versions of services in IST’s system?

A couple of pros and cons regarding the implementation were identified and exemplified. A set of rules were also outlined that are required to follow if the implementation is to be used in their current system.

RQ2: How well would the implementation work with Continuous Delivery?

The implementation was compared towards existing research for CD. Any identified problems about adopting CD from previous research that were relatable to the implementation were exemplified and presented as to how they could be managed by the implementation, problematic or if it was a mix of both.

The implementation mainly resides in the registrator application IST has built which is based off the one developed by Glider Labs [14]. Thus anyone who also uses a similar registrator application should be able to relate to the implementation when registering their services, but to modify the endpoint relies completely on using Nginx.

After the comparison towards CD in this study, I personally believe that the best approach for adopting CD would be to compare towards existing research and cases, and possibly receive opinions from people with experience from prior adoption cases. This is because a lot depends on an organisation as a whole to adopt CD so any reference towards similar organisations or processes that have gone through the adoption process should give good points of reference for any possible problems. This study just compared the implementation and its functionality towards CD and there were still a lot of relatable issues. There are still a lot of other processes that needs to be looked into that have known possible problems for adopting CD (e.g. human and organisational).
6.1 Future Research

For future research there was a feature IST mentioned that was not investigated in this study, namely the possibility to do an incremental increase of use for a new version. To exemplify this, imagine BlueGreen deployment where the blue version is the current stable version. When the new green version would be released only a certain amount of users would end up using that version, e.g. at release maybe 5% would use the new green and the rest would use the stable blue version. This percentage amount would then incrementally increase over time unless the version is taken down due to discovered bugs or issues. This would result in less amount of users encountering possible bugs or issues and giving an overall better user experience while still being able to provide the benefits of BlueGreen deployment. This was something that could have been investigated with this implementation by regarding one version as stable and another one as new and then having the service registered under the group name work to split the requests as described above.

Regarding CD, more adoption cases should still be researched until a stable generic process has been established to adopt CD (if one will ever arise). Trying to predict issues without prior experience is too riskful of an approach and thus any possible reference to problems could be vital for an organisation when adopting CD.
References


